DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

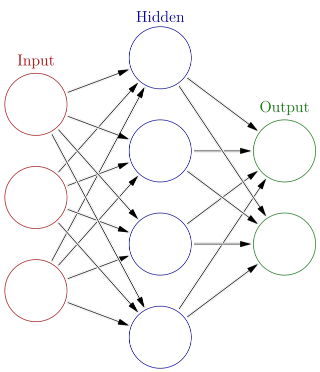
UNIVERSITÀ DEGLI STUDI DI PADOVA

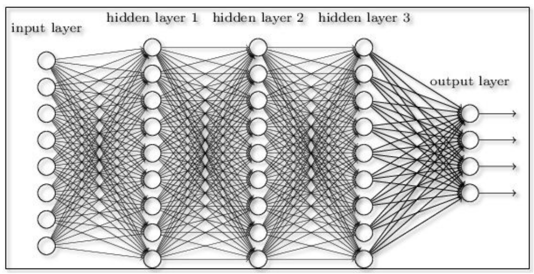
Convolutional Neural Networks

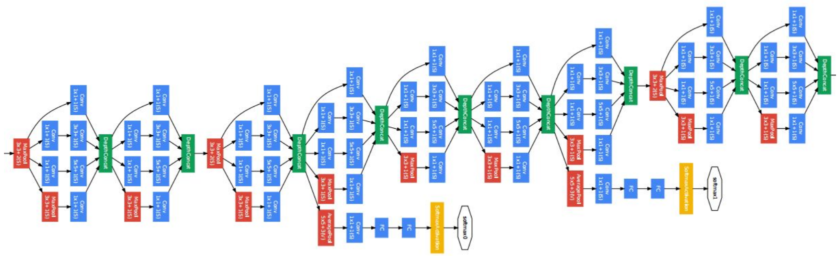
Machine Learning 2022-23

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Recall: Artificial Neural Networks



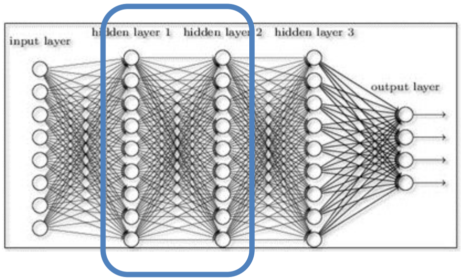




- Model of computation inspired by the structure of neural networks in the brain
- Large number of basic computing devices (neurons) connected to each other
- Represented with directed graphs where the nodes are the neurons and the edges corresponds to the links between the neurons
- Proposed in 1940-50
- First practical applications in the 80-90 but practical results were lower than SVM and other techniques
- **From 2010 on deep architectures with impressive performances**

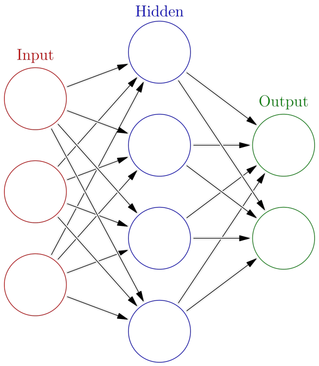
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Recall: Feedforward NN



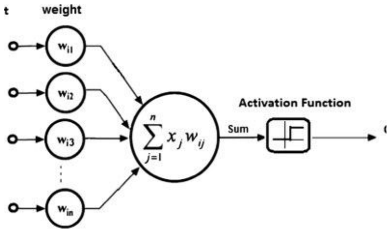
Feedforward network: the graph has no edges

It is typically organized into layers: each neuron takes in input the output of all neurons from the previous layer



Notation: NN: G=(V,E)

- V: neurons |V|: size of the network
- E: connection between neurons (directed edges)
- $w: E \rightarrow \mathbb{R}$ weight function over the edges



Each neuron:

1. Takes in input the sum of the outputs of the connected neurons weighted by the edge weights
2. Applies to it a simple scalar function (activation function, σ)

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Recall: NN Training Algorithm

BackPropagation algorithm with SGD

Input: training data $(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m)$

Output: NN weights $w_{ij}^{(t)}$

Initialize $w_{ij}^{(t)}$ $, \forall i, j, t;$

for $s \leftarrow 0,1,2, \dots$ **do** // until convergence

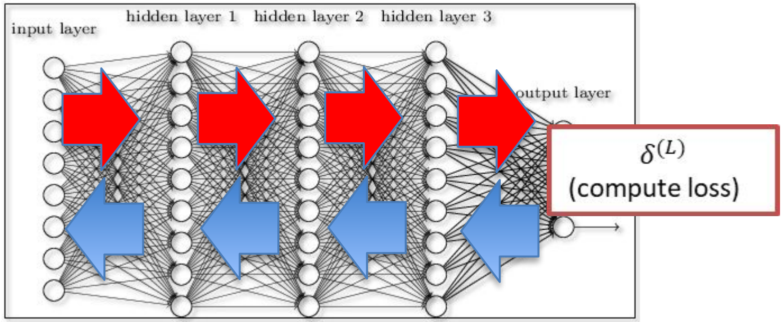
pick $(\boldsymbol{x}_k, y_k)$ at random from training data; // SGD

compute $v_{t,j}$ $, \forall j, t$ // forward propagation

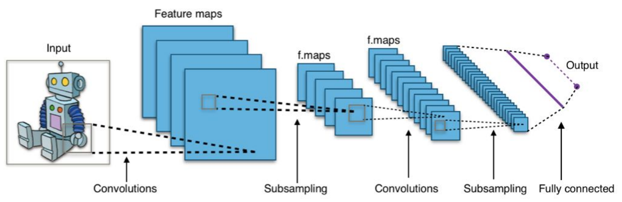
compute $\delta_j^{(t)}$ $, \forall j, t$ // backward propagation

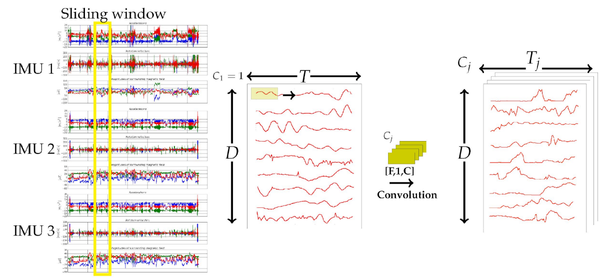
$w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta v_{t-1,i} \delta_j^{(t)}$ $\forall i, j, t$ // update weights

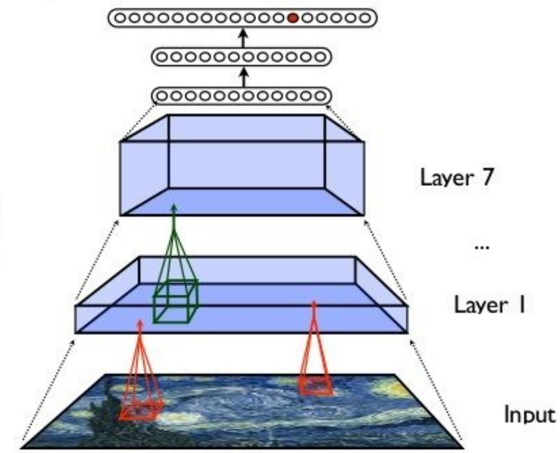
if *converged* **then return** $w_{ij}^{(t)}$ $, \forall i, j, t;$



Issues of Fully Connected Feedforward Networks







Two main issues in the NN model we have seen up to now

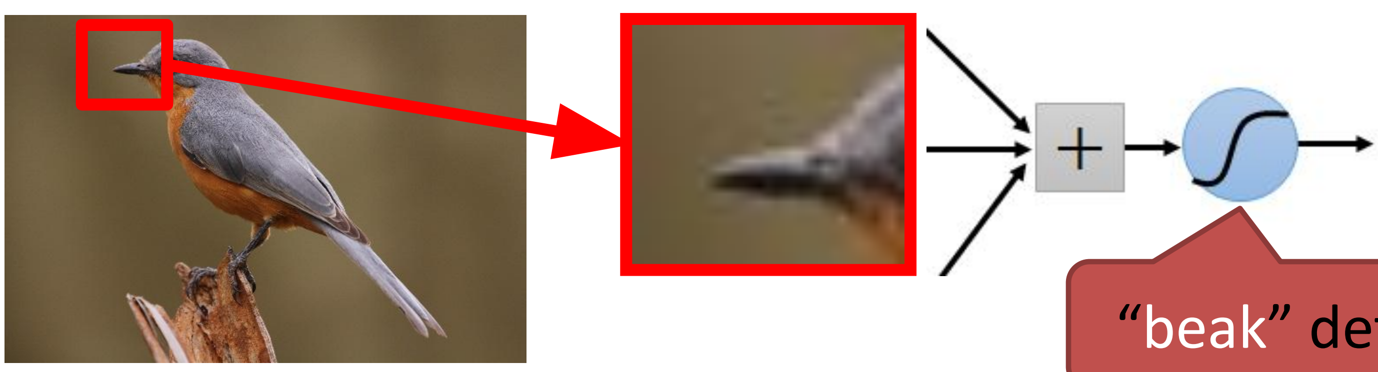
1. Each neuron of layer *t-1* connected with each neuron of layer *t*
   → *huge number of edges/weights (quadratic w.r.t. number of neurons)*
2. The *domain structure is not taken into account*
   - The model does not consider that a neuron can be "*closer*" (→ more related) to some neurons and less to others
   - Some domains have a structure
     - E.g., grid of pixels in an image, sequence of samples in an audio signal, letters of a word in a text, ...
     - Need to capture the fact that a pixel in an image is more related to the close pixels than to the far apart ones or a letter in a text is more related to letters of the same word than to the ones 10 pages ahead !
   - Interesting features are often local, shift-invariant and deformation-invariant
   - By simply placing data in a vector → loose spatial or temporal structure

→ Need to update the NN model

Example: Learning an Image (1)

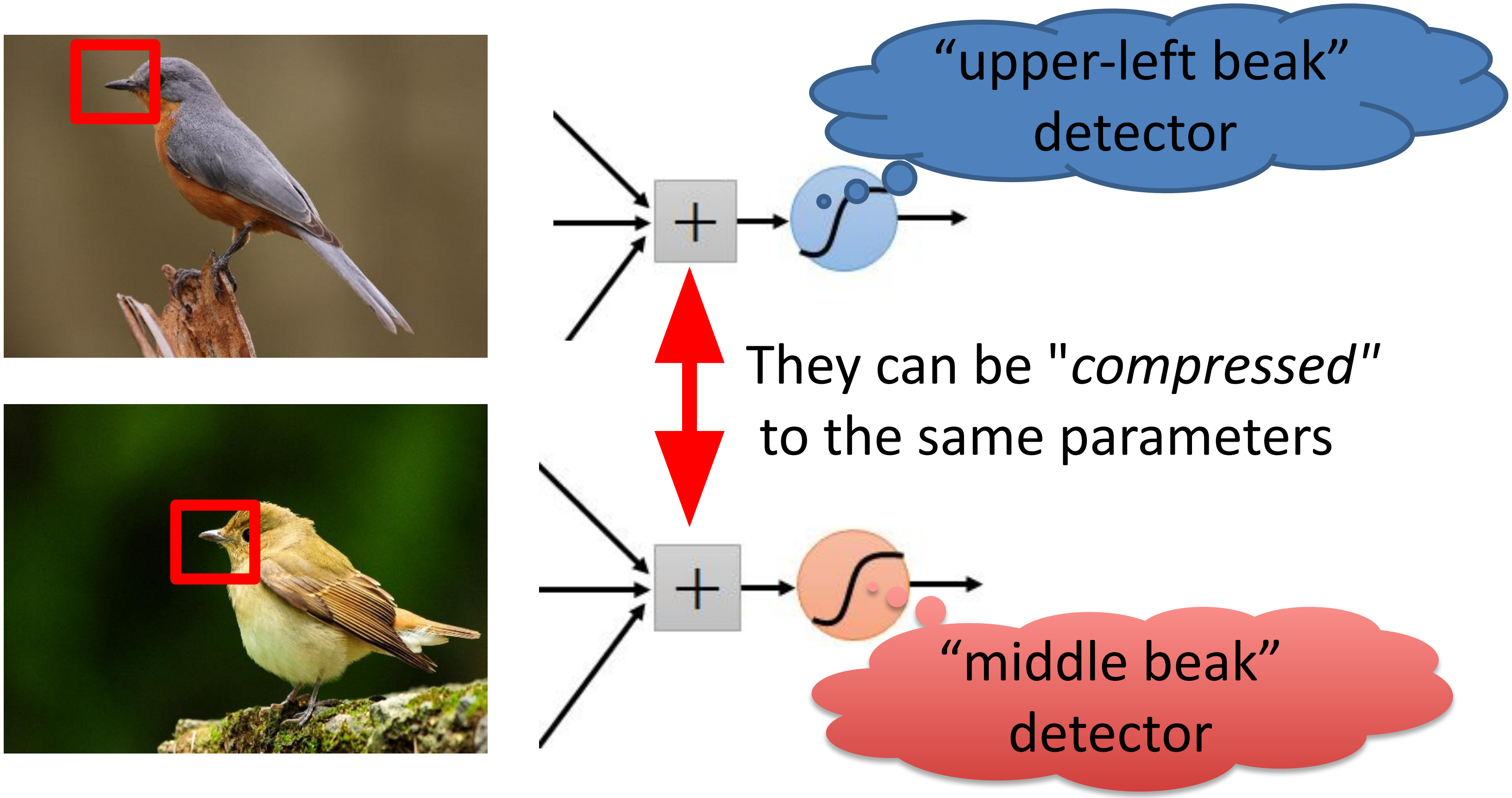
Some patterns are much smaller than the whole image

Can represent a small region with fewer parameters



"beak" detector

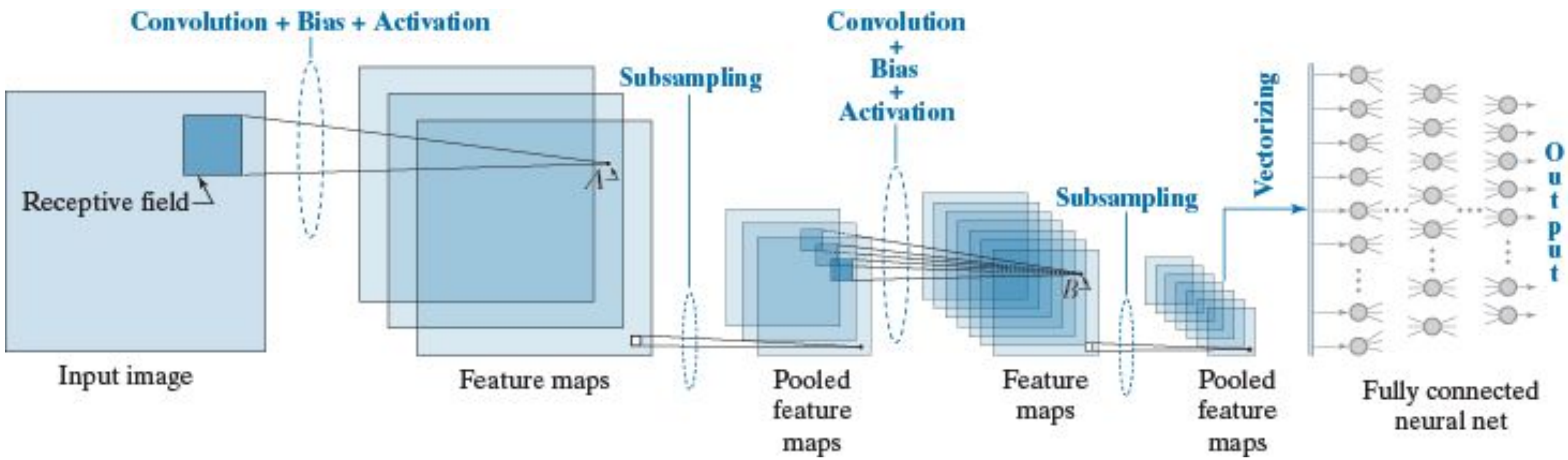
Example: Learning an Image (2)



- The same pattern can appear in different places
- Similar detectors in different regions share similar parameters
- What about training some "small" detectors and let each detector "move around" ?

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

From NNs to Convolutional Neural Networks

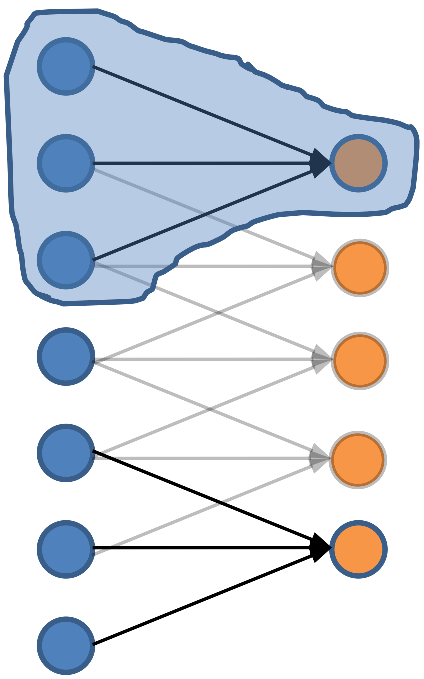


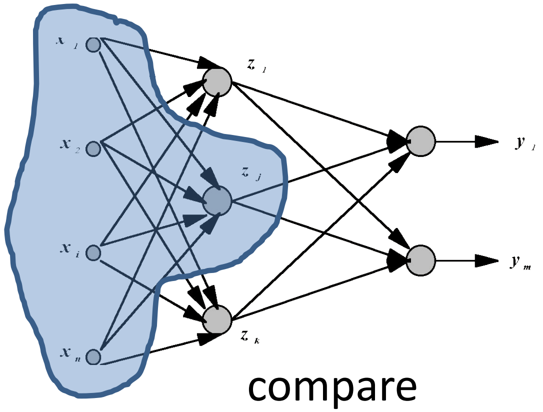
Convolutional Neural Network (CNN)

1. *Local connectivity*: receptive field for each neuron
2. *Shared ("tied") weights*: spatially invariant response
3. *Multiple feature maps*
4. Subsampling (*pooling*)

Convolutional NNs

1. Local connectivity



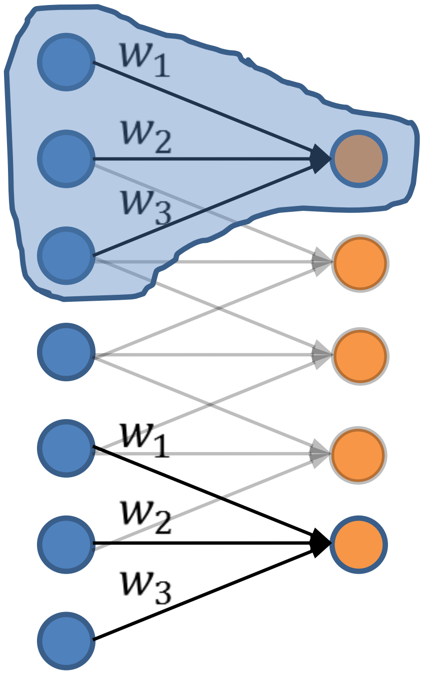


compare

- Each orange unit is only connected to **neighboring** blue units

Convolutional NNs

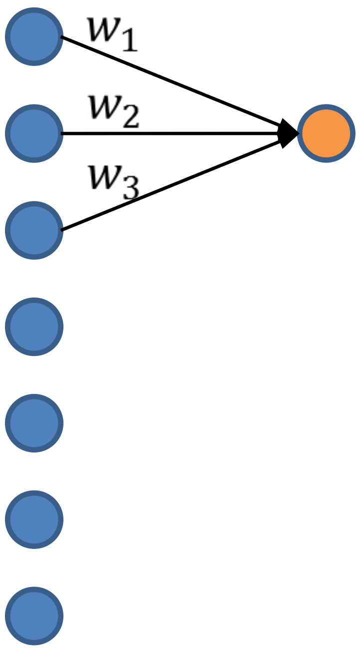
2. Shared ("tied") weights



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

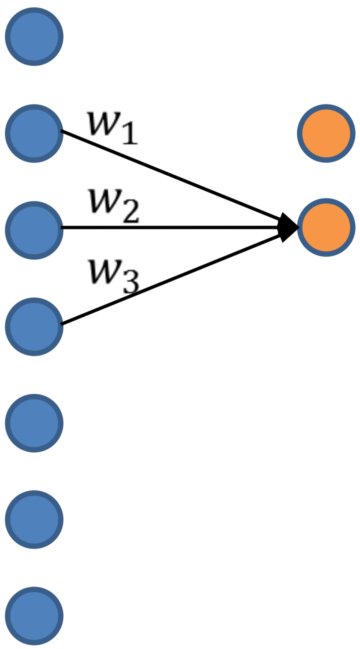
- Convolution with 1-D filter: $[w_1, w_2, w_3]$



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

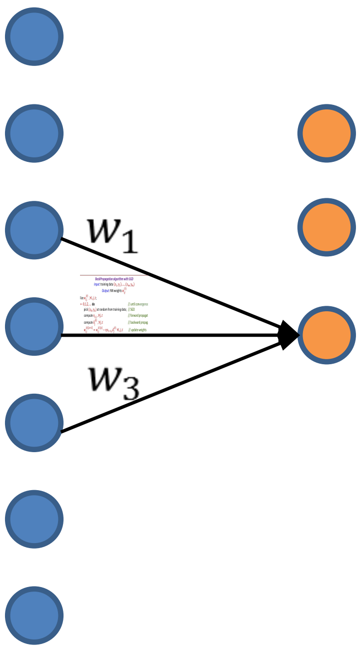
- Convolution with 1-D filter: $[w_1, w_2, w_3]$



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

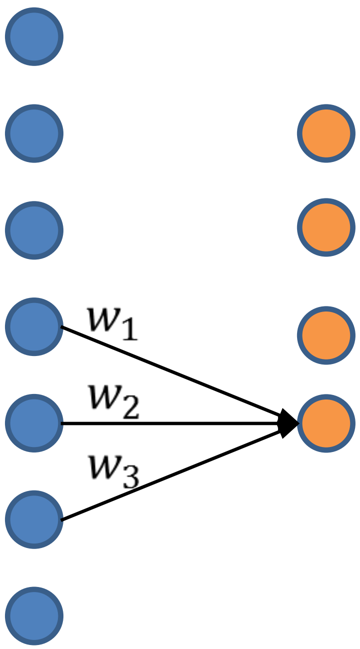
- Convolution with 1-D filter: $[w_1, w_2, w_3]$



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

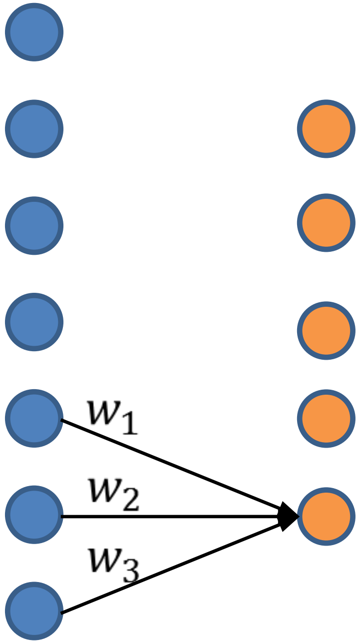
- Convolution with 1-D filter: $[w_1, w_2, w_3]$



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

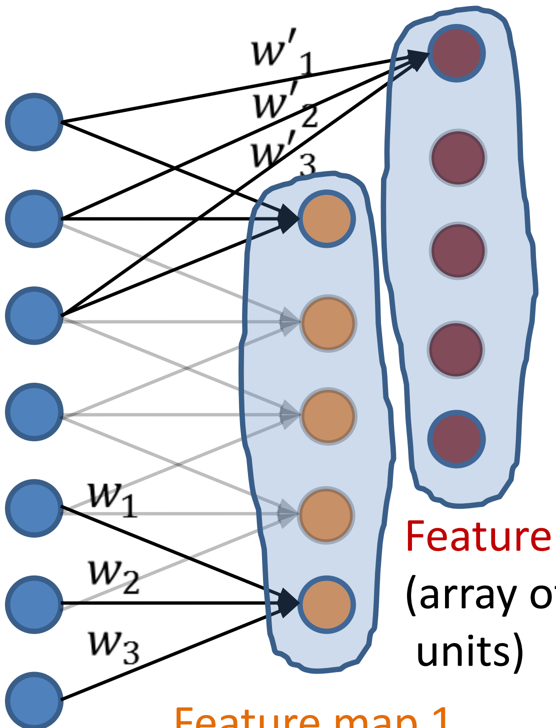
- Convolution with 1-D filter: $[w_1, w_2, w_3]$



- All orange units **share** the same parameters $\boldsymbol{w}$
- Each orange unit computes the **same function,** but with a **different input window**

Convolutional NNs

3. Multiple feature maps



Feature map 2
(array of red units)

Feature map 1
(array of orange units)

- All orange units compute the **same function** but with a **different input windows**
- Orange and red units **compute different functions**

Convolution

?	?	?				
?	1	0	0	0	0	1
?	0	1	0	0	1	0
	0	0	1	1	0	0
	1	0	0	0	1	0
	0	1	0	0	1	0
	0	0	1	0	1	0

6 x 6 matrix

Convolution at boundaries:
- Stop before → reduced output size
- Use padding to extend input size

These are the network parameters to be learned.

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1
(feature map 1)

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2
(feature map 2)

⋮ ⋮

Each filter detects a small pattern (3 x 3)

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

Dot product

3 -1

6 x 6 matrix

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 matrix

3 -3

Stride: allows the filter to move in steps of multiple samples (alternative to pooling to reduce resolution)

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 matrix

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Convolution

-1	1	-1
-1	1	-1
-1	1	-1

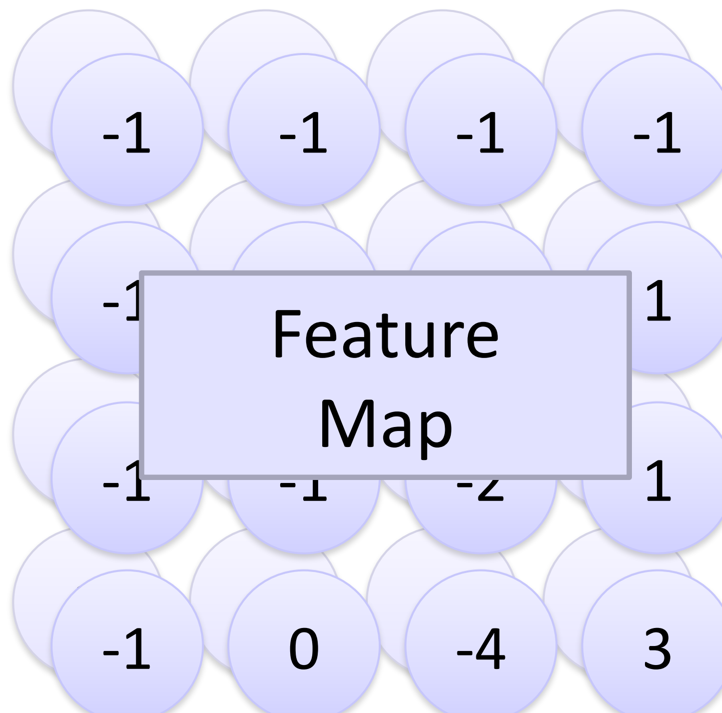
Filter 2

stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

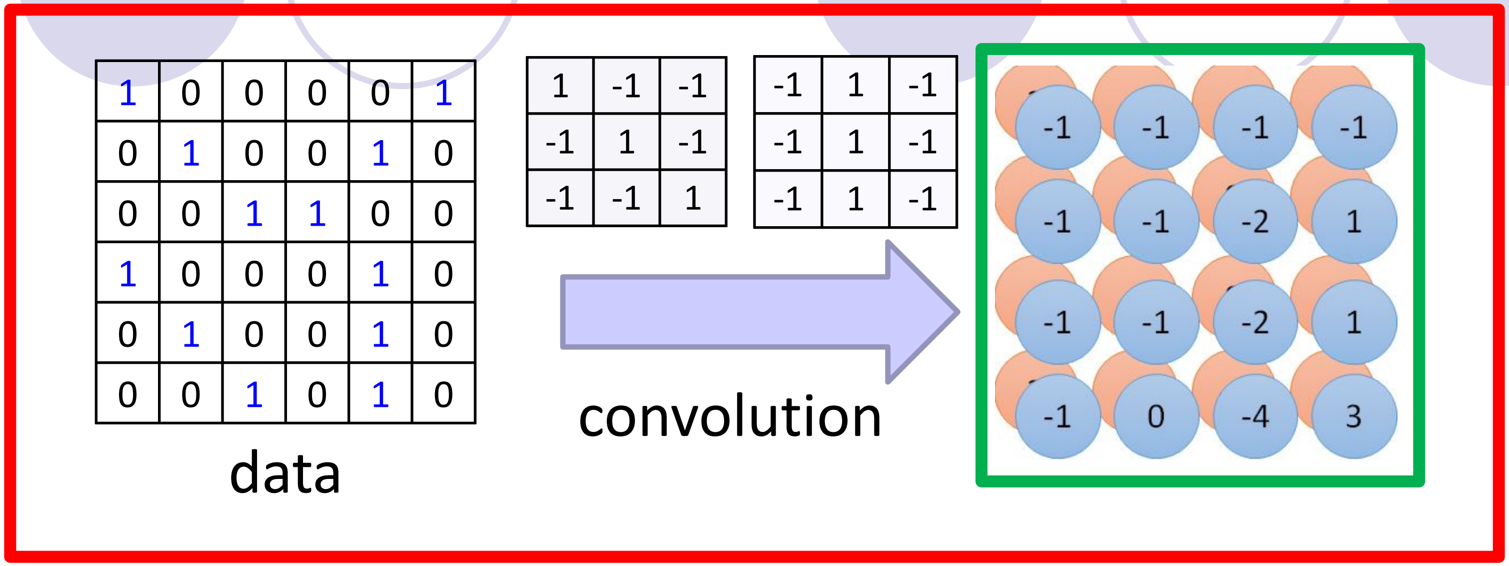
6 x 6 matrix

Repeat this for each filter

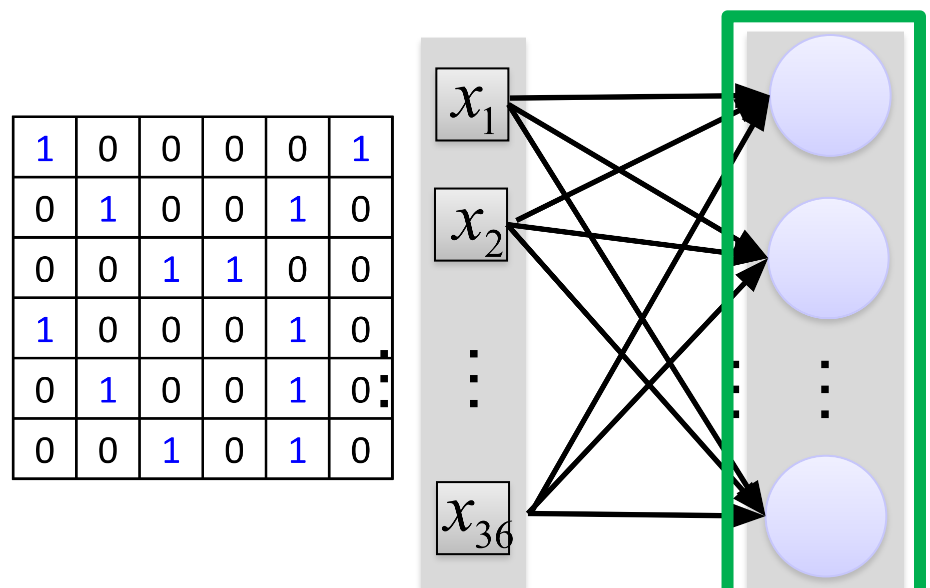


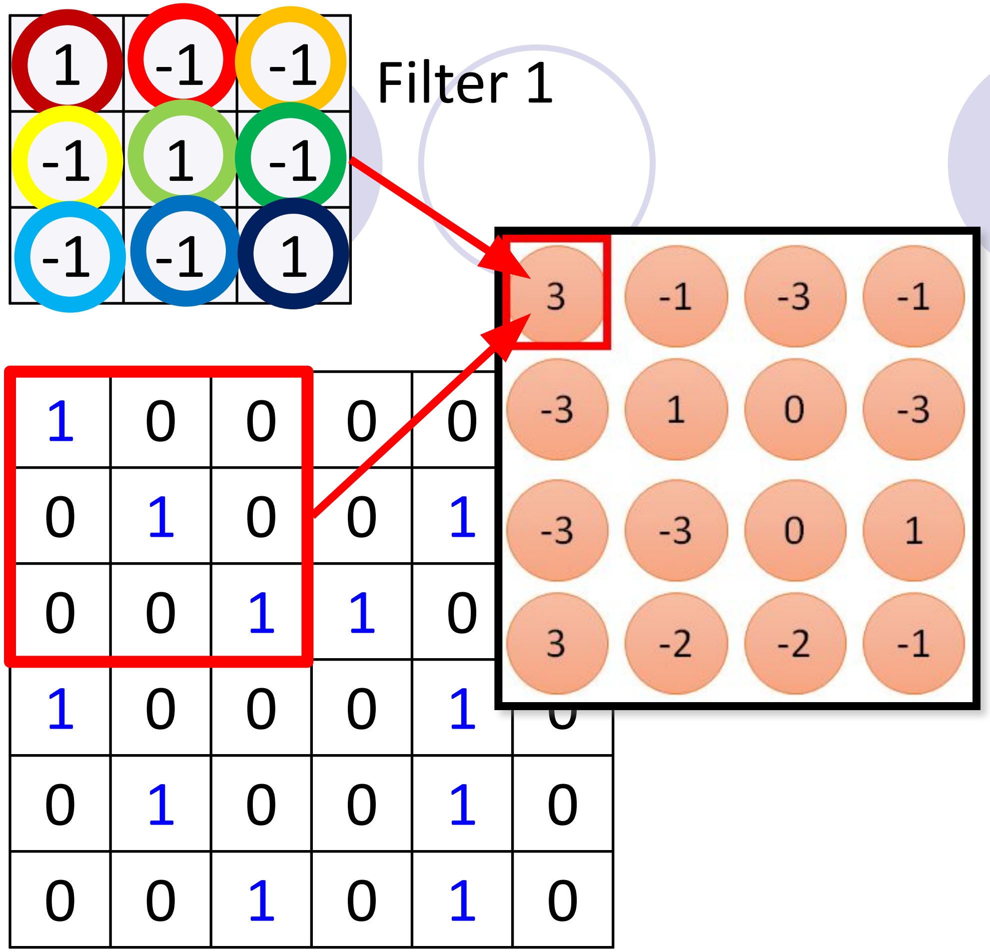
Two 4 x 4 images
Forming 2 x 4 x 4 matrix

Convolution v.s. Fully Connected



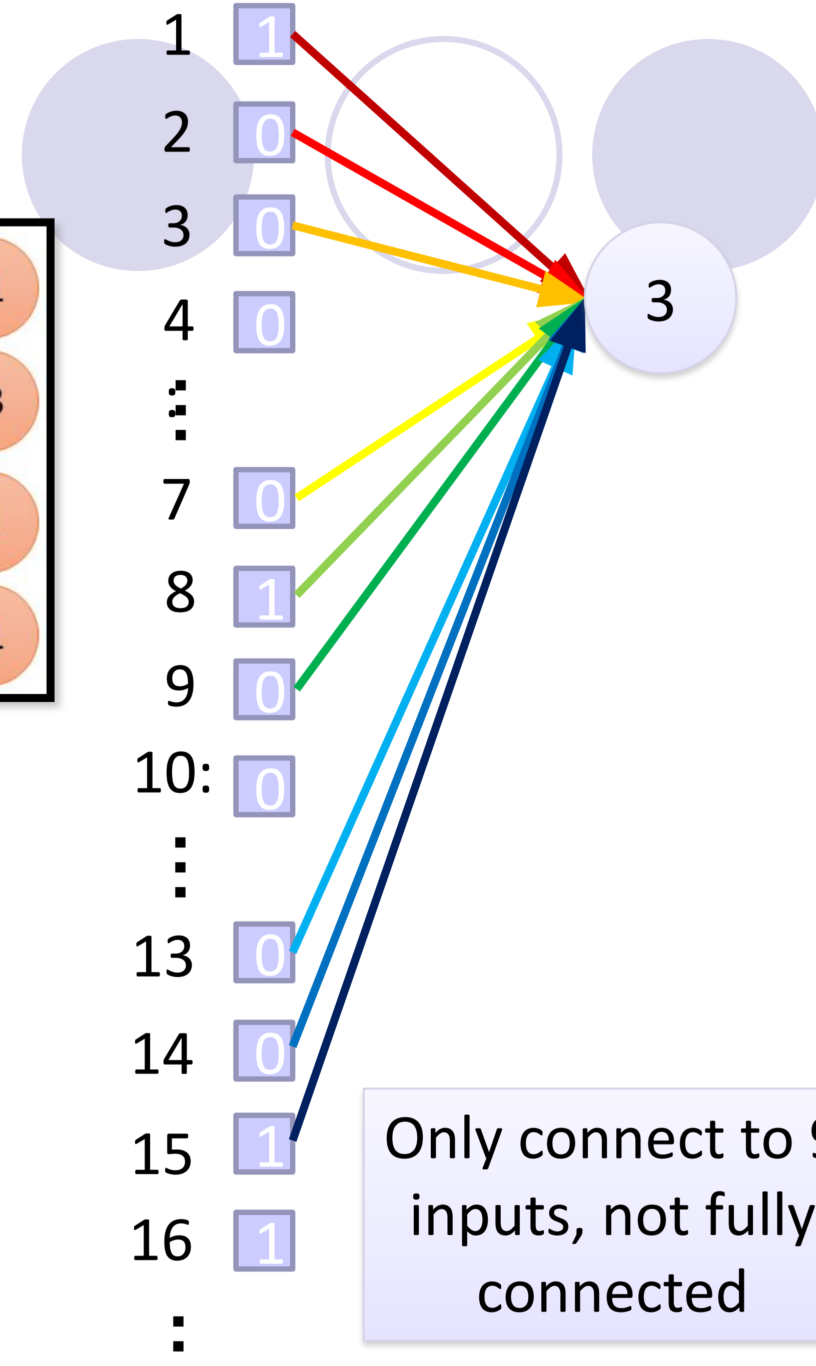
Fully-connected



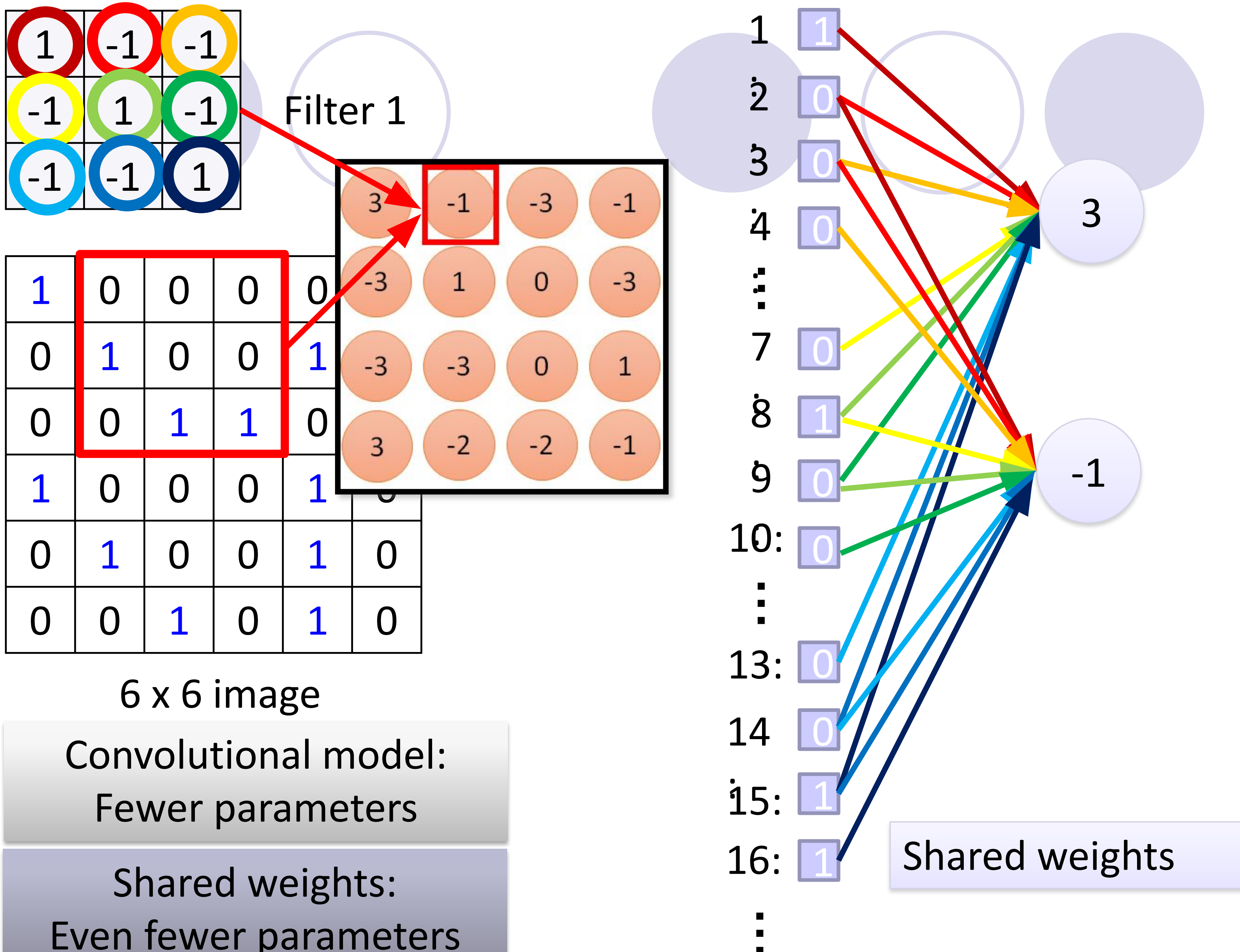


6 x 6 image

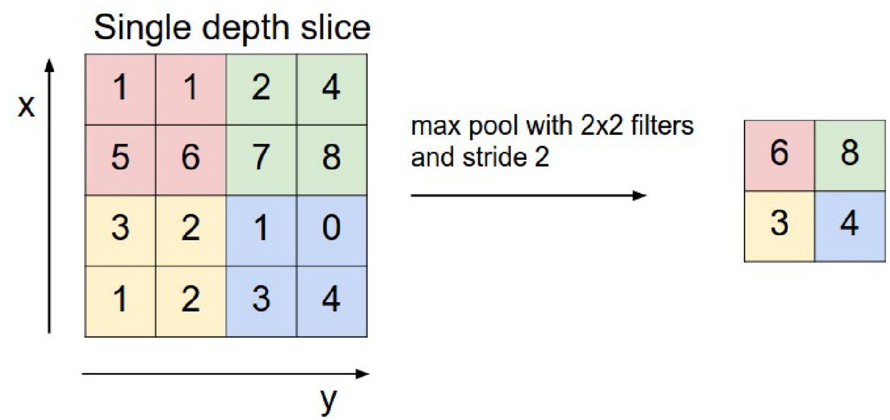
Convolutional model: fewer parameters!



Only connect to 9 inputs, not fully connected


Filter 1
6 x 6 image
Convolutional model:
Fewer parameters
Shared weights:
Even fewer parameters
1
2
3
4
7
8
9
10:
13:
14
15:
16:
3
-1
Shared weights

Pooling Layer



- Reduce resolution→next convolutional layer is applied at a larger scale
- Originally introduced to reduce the computational burden and the memory requirements…
- …but turned out to be crucial to improve performance in many applications since it increases the receptive field of the inner layers
- Adds some deformation invariance too
- *Max Pooling* is the most common example of such layer: it works very well, it is quick, and can be efficiently implemented in hardware

Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

Why Pooling ?

Subsampling pixels will not change the object

bird

Subsampling

bird

We can subsample the pixels to make image smaller and use fewer parameters to characterize the image
However this is not the only reason for using pooling…

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Pooling and Receptive Size

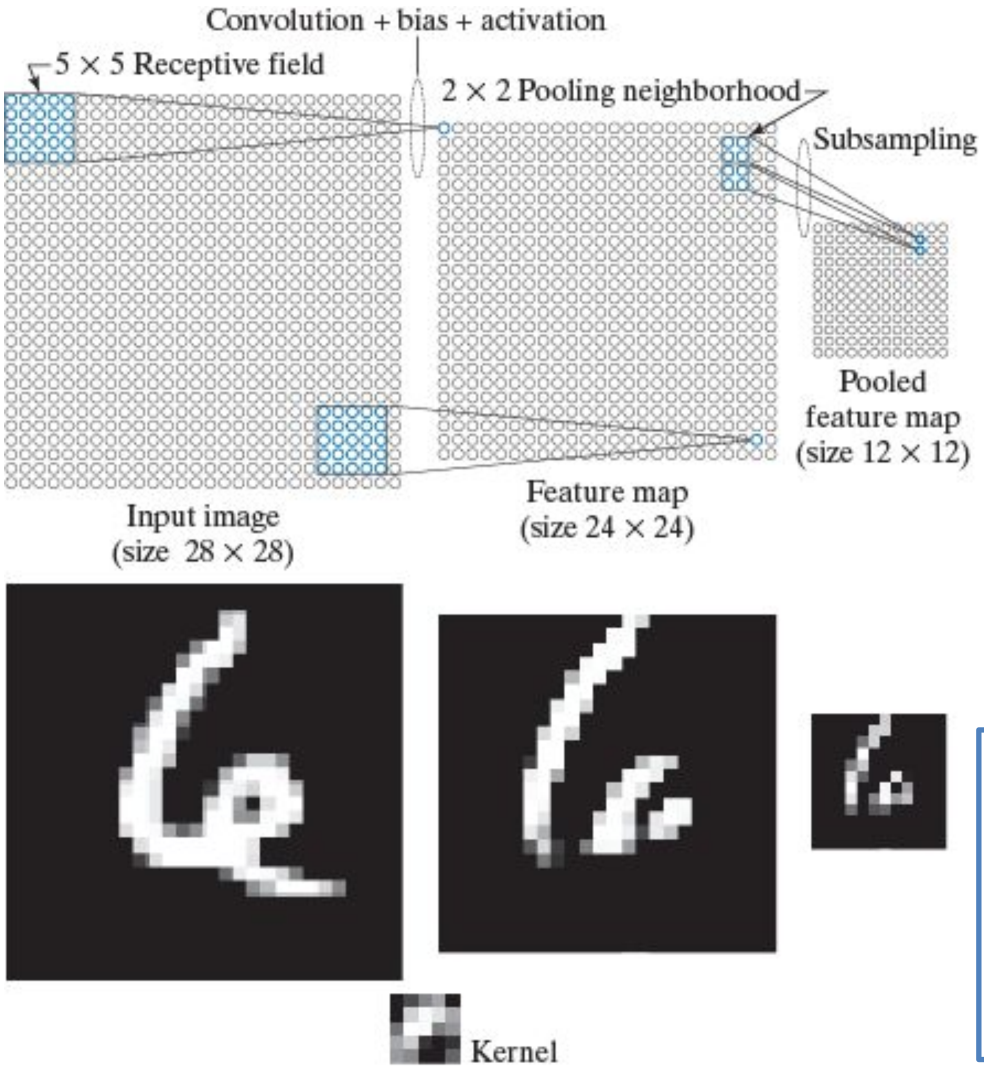
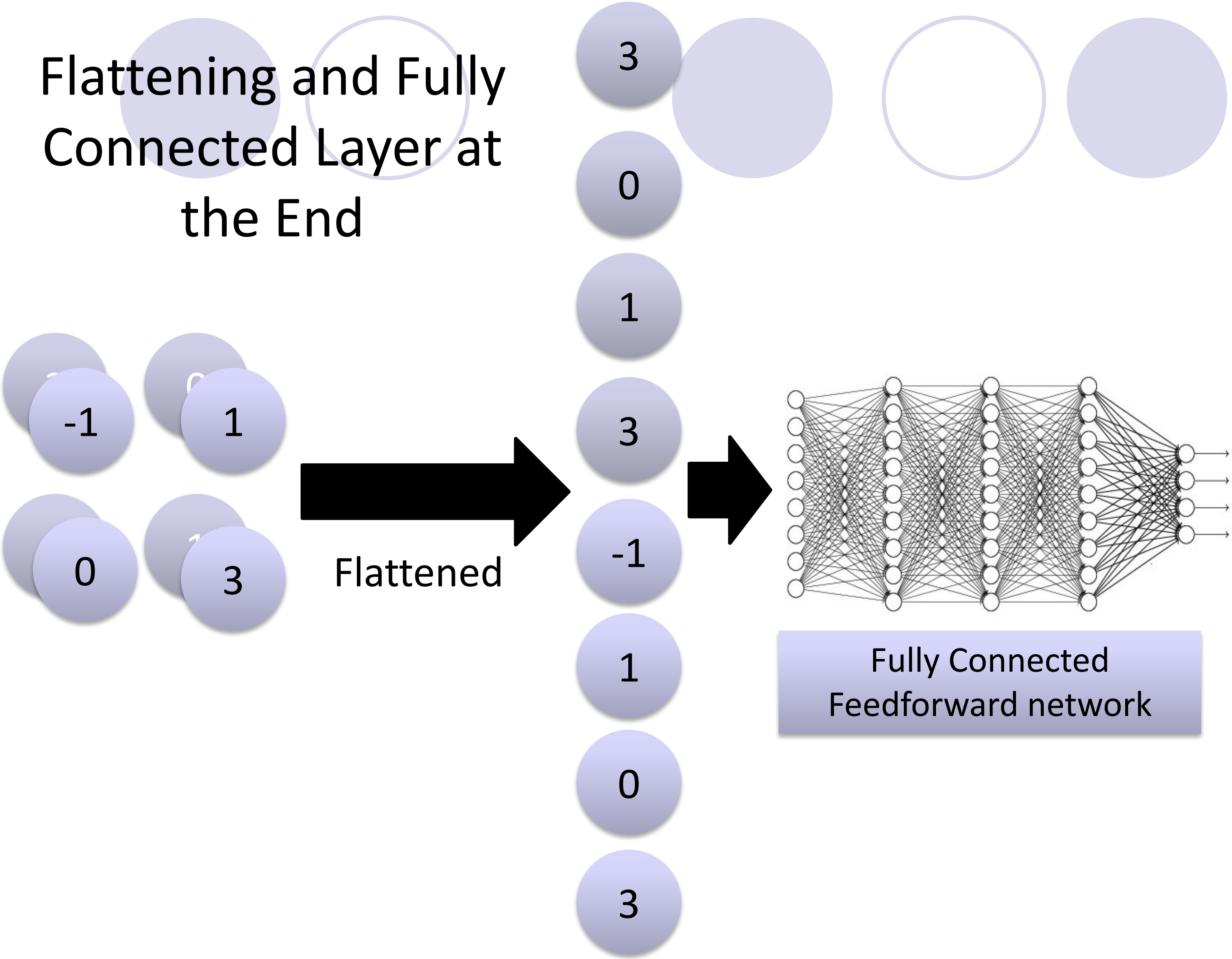


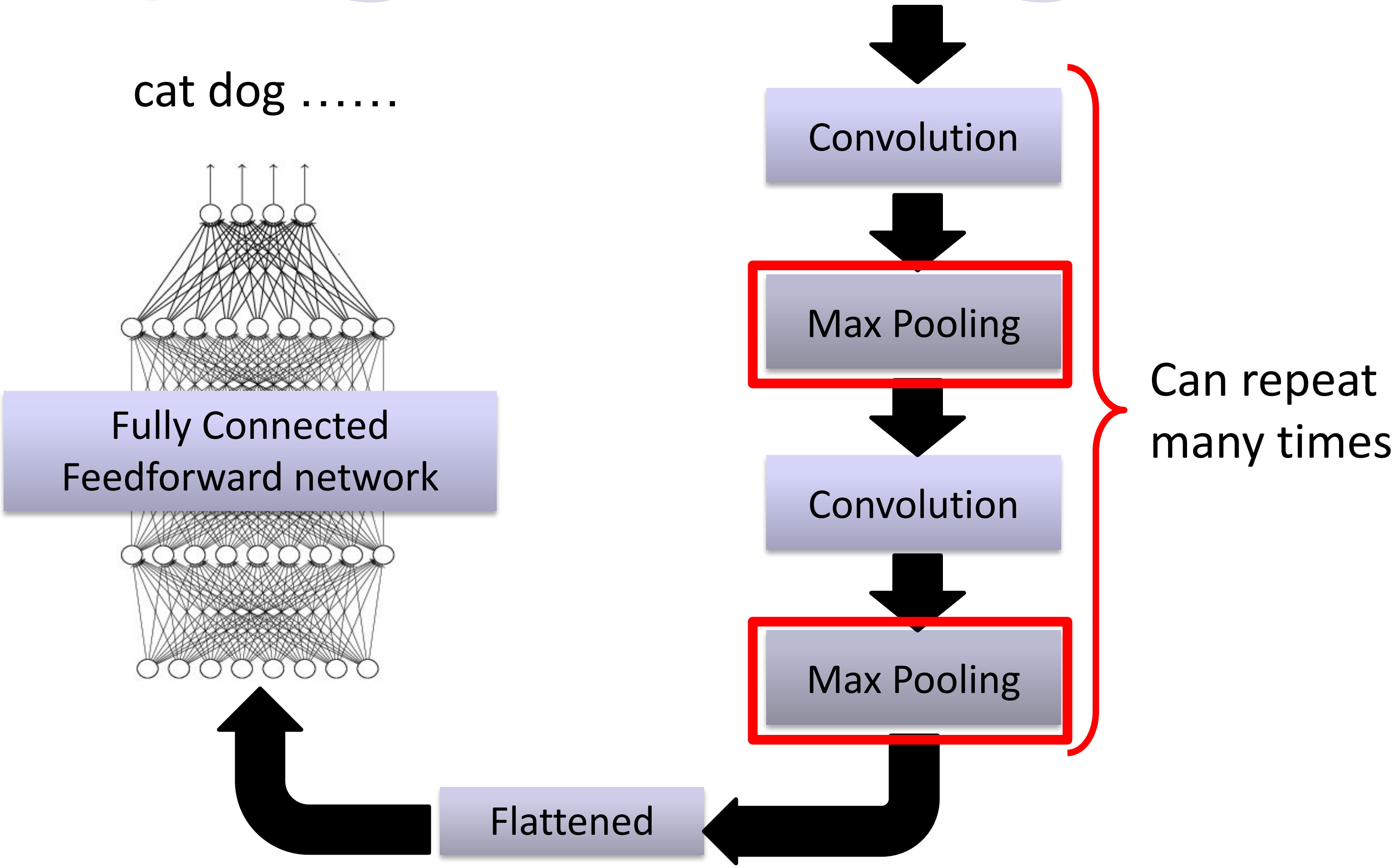
FIGURE 12.41 Top row: How the sizes of receptive fields and pooling neighborhoods affect the sizes of feature maps and pooled feature maps. Bottom row: An image example. This figure is explained in more detail in Example 12.17. (Image courtesy of NIST.)

A small convolution window after pooling corresponds to a larger area in previous layers

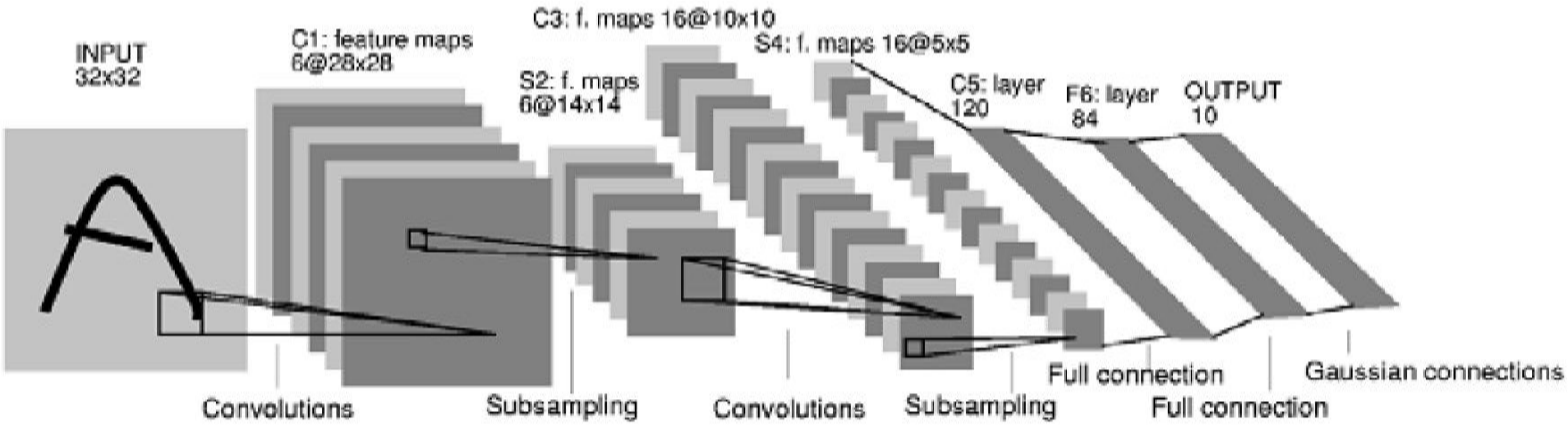
Flattening and Fully Connected Layer at the End



Baseline CNN model



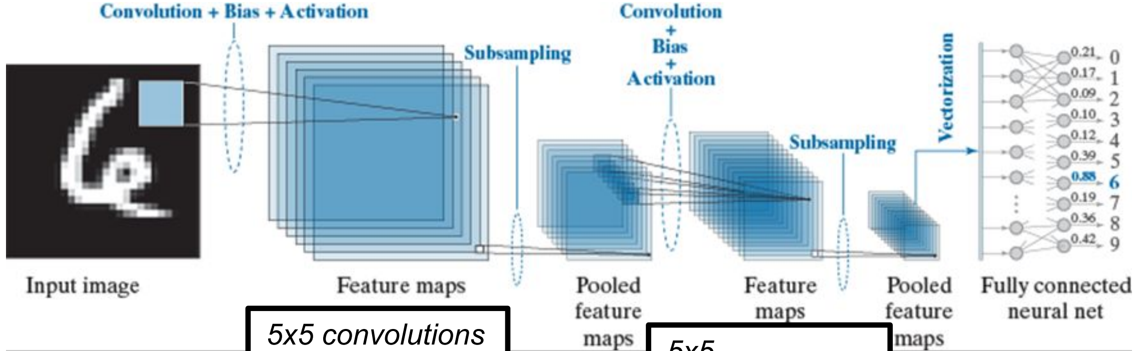
Convolutional Networks



- *Hierarchical representation*: low level features in the first layer, then moving to higher and higher abstraction levels
- *Weight sharing*: huge reduction of complexity w.r.t. a fully connected network
- The CNN model "*compresses*" a fully connected network in various ways:
  - Reducing the number of connections
  - Shared weights on the edges
  - Max pooling further reduces the complexity

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Example: Simple CNN



5x5 convolutions
6 feature maps
2x2 pooling

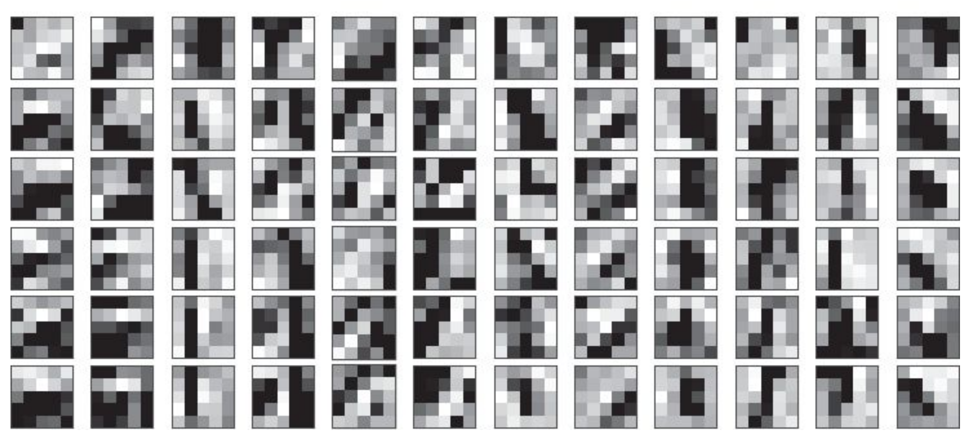
(25+1)*6=156 params

5x5
convolutions
12 feature
maps
2x2 pooling

(6*25+1)*12=1812 params

Layer 1

Layer 2



Example: Feature Maps

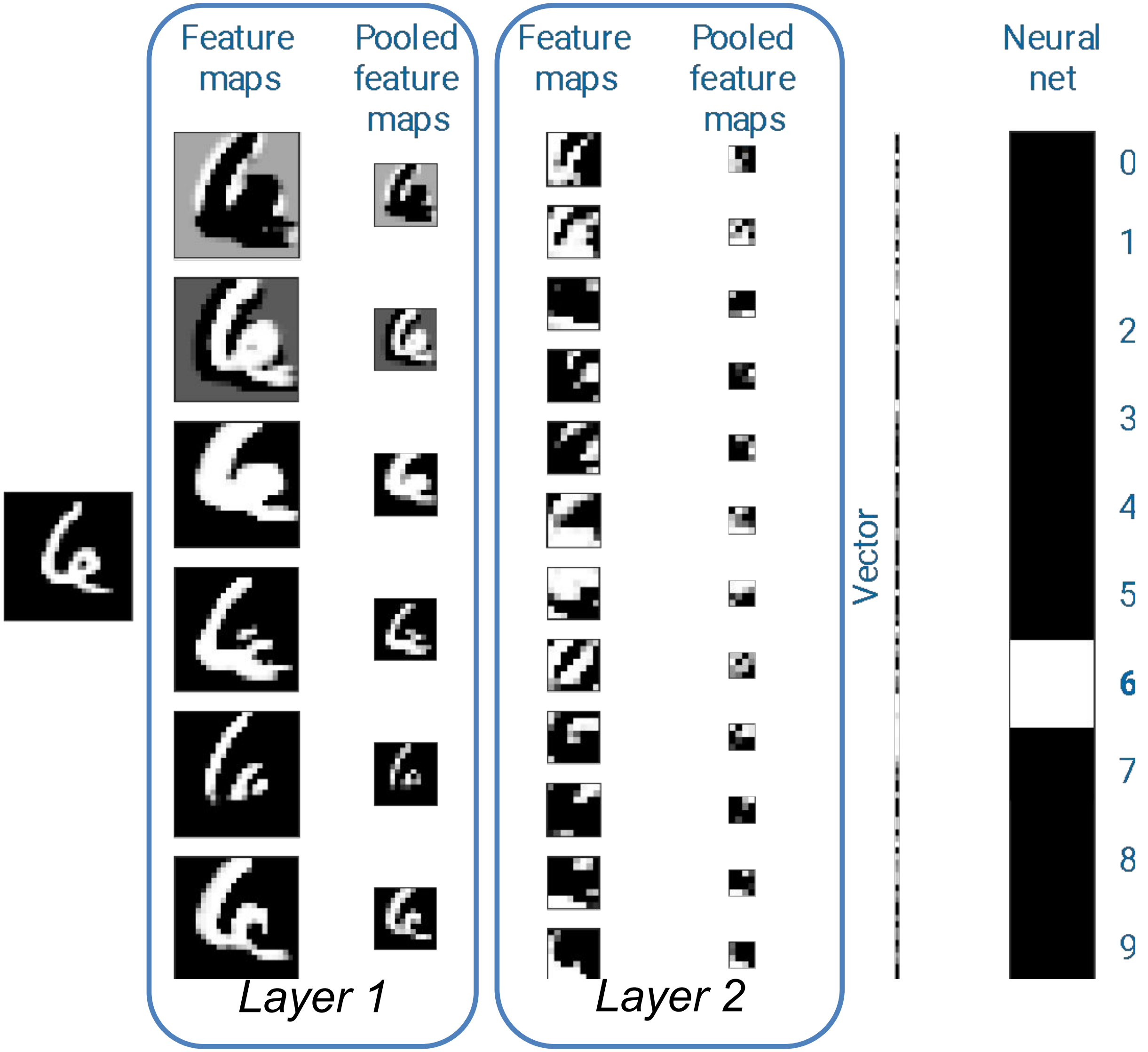


FIGURE 12.44 Visual summary of an input image propagating through the CNN in Fig. 12.42. Shown as images are all the results of convolution (feature maps) and pooling (pooled feature maps) for both layers of the network. (Example 12.17 contains more details about this figure.)

Example: AlphaGo

→ Neural Network → Next move

(19 x 19 positions)

Black: 1

white: -1

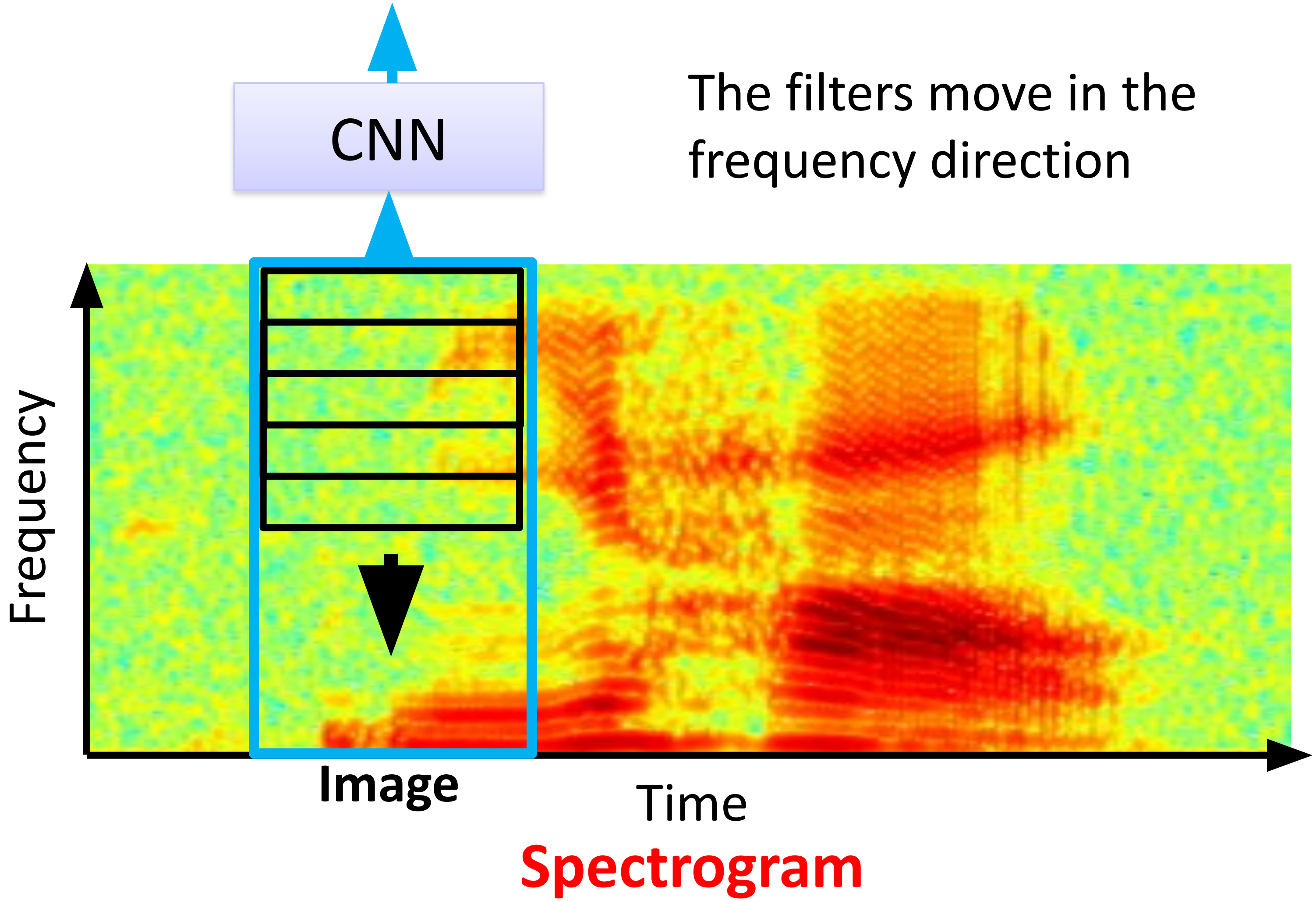
none: 0

Fully-connected feedforward network can be used

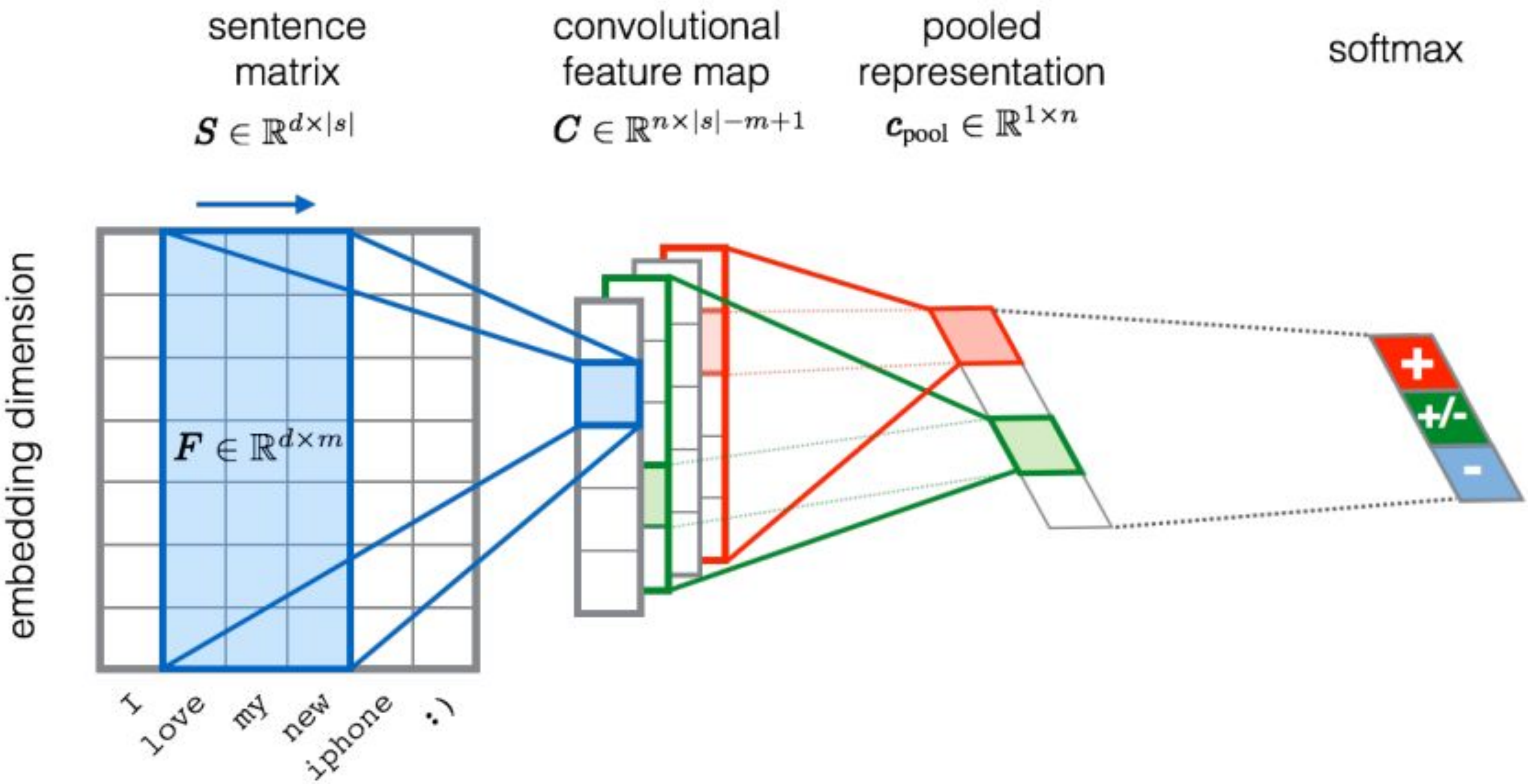
But CNN performs much better

Example: CNNs in Speech Recognition

The filters move in the frequency direction

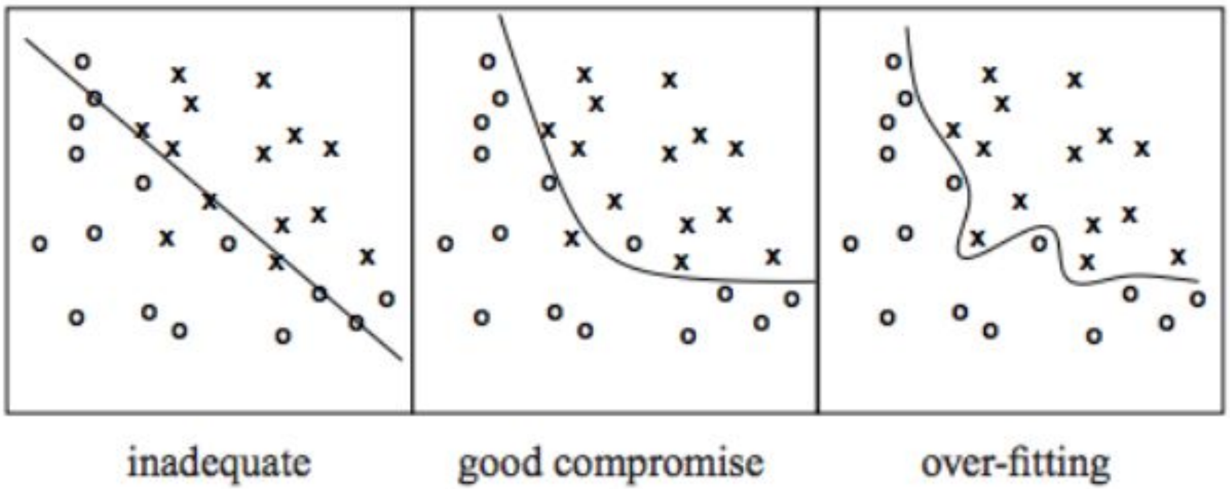


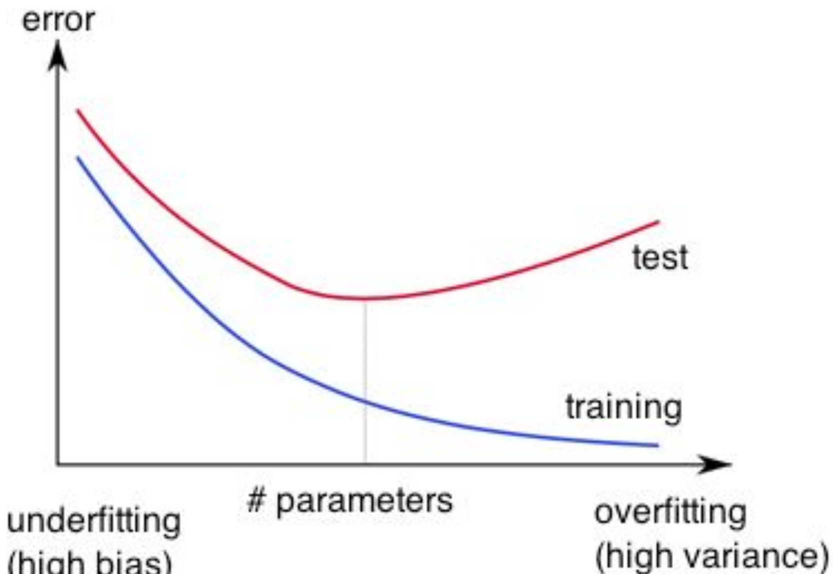
Example: CNNs in text classification



Source of image:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.703.6858&rep=rep1&type=pdf

Avoid Overfitting



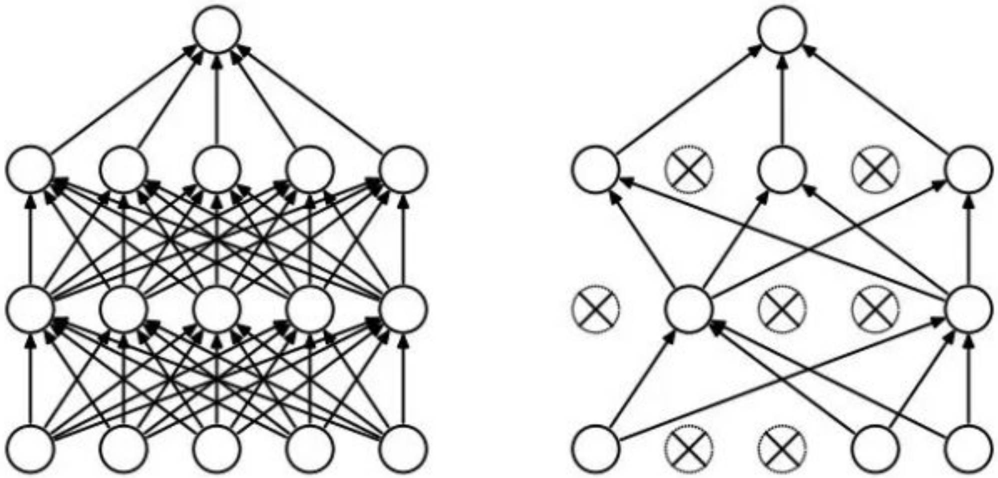


Learned hypothesis may fit the training data very well, even outliers (noise), but fail to generalize to new examples (test data)

- *Do not use a too complex network if training data is limited*
- *Various techniques can be used to deal with this problem*

http://wiki.bethanycrane.com/overfitting-of-data *https://www.neuraldesigner.com/images/learning/selection_error.svg*

Avoid Overfitting: Dropout



Dropout

- Randomly drop neurons (along with their connections) during training
- Each unit retained with fixed probability *p*, independent of other units
- Hyper-parameter p to be chosen (tuned)
- At each step the network is trained with only a subset of the neurons
- Avoid that the output depends "too much" on a single neuron
- Typically applied only to some layers (e.g., fully connected at the end)
- More stable / less risk of overfitting

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." Journal of machine learning research (2014)

Avoid Overfitting: Regularization

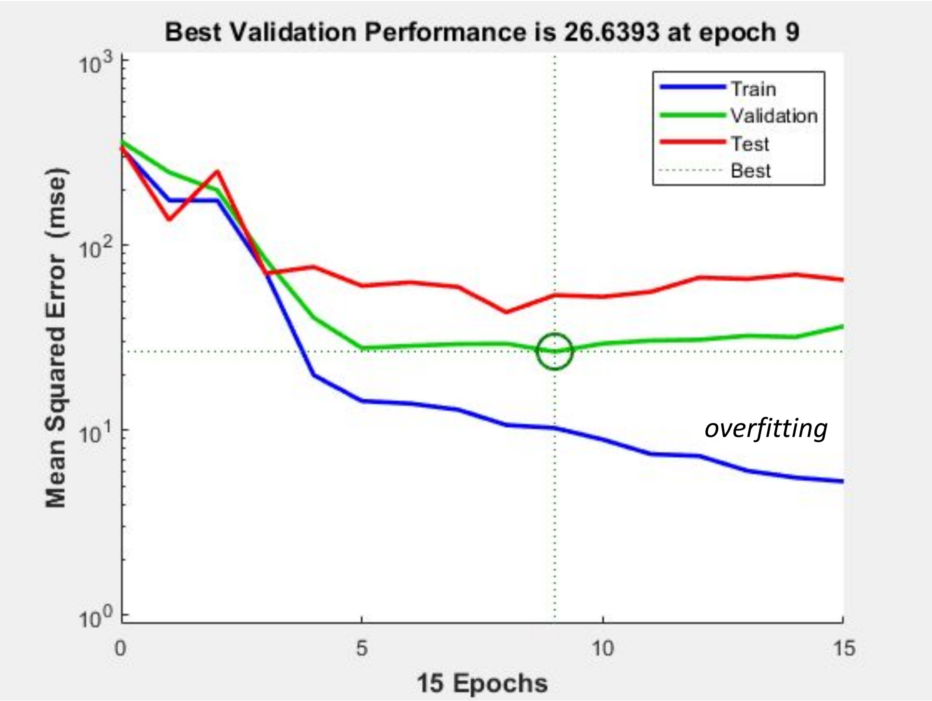
$$J_{reg}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{m}\sum_{i,j,t}\left|\boldsymbol{w}_{ij}^{(t)}\right|$$

$$J_{reg}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{m}\sum_{i,j,t}\left(\boldsymbol{w}_{ij}^{(t)}\right)^2$$

Regularization

- Regularization term added to the loss function
- Penalizes big weights and reduces risk of overfitting
- Regularization parameter λ determines how relevant regularization is during gradient computation
- Big λ → big penalty for big weights
- L1 or L2 regularization can be used

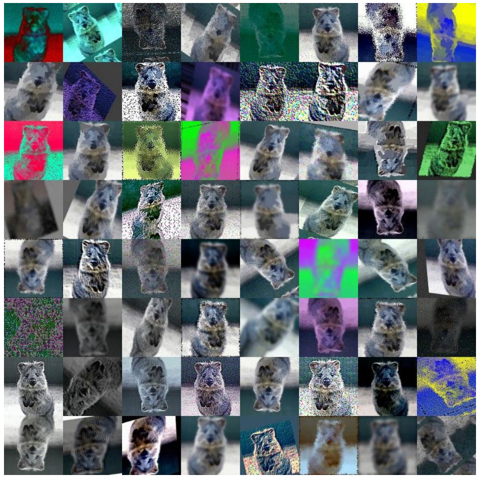
Avoid Overfitting: Early Stopping



Early-stopping

- Use validation error to decide when to stop training
- Stop when monitored loss has not improved after *n* subsequent epochs
- Parameter "*n*" is called patience

Avoid Overfitting: Data Augmentation



Data Augmentation

- Add a little bit of variance to the data to "virtually" increase number of training samples (but new samples are correlated, not the same as having more samples)
- Artificially add noise
- Apply random transformations (depend on data type)
  - Crop part of the data
  - Resize/rescale data
  - Rotate
  - Custom transformations depending on data type (e.g., for images: flip horizontally, adjust hue, contrast and saturation)

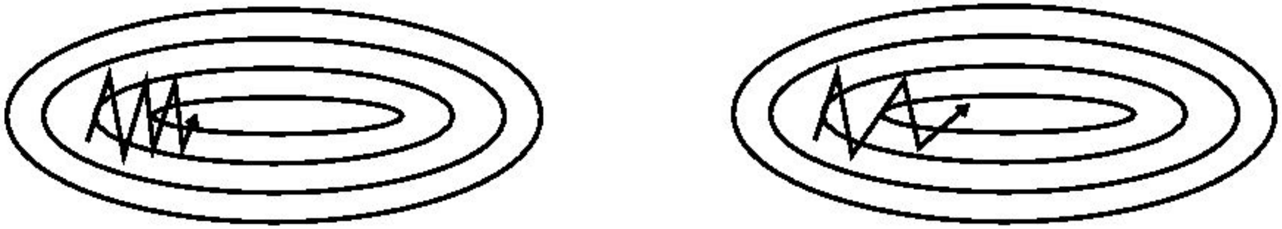
Gradient Descent for NN (1)

- Huge number of parameters, very challenging optimization
- A variety of optimization algorithms have been proposed (recall GD lecture)

1. *Basic Gradient Descent (GD)*
    - Computes the gradient of the cost function w.r.t. to the parameters for the entire training dataset
    - Need to calculate the gradients for the whole dataset to perform just one update
    - Can be very slow and is intractable for datasets that don't fit in memory
2. *Stochastic Gradient Descent (SGD)*
    - Performs a parameter update for each training example
    - It is usually much faster but performs frequent updates with a high variance and can be unstable
    - SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima
    - On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Gradient Descent for NN (2)



3. *Mini-batch gradient descent:*
   - Compromise between GD and SGD: performs an update for every mini-batch of *n* training examples
   - Reduces the variance of the parameter updates, which can lead to more stable convergence
   - Can make use of highly optimized matrix computations in state-of-the-art deep learning libraries
   - Common mini-batch sizes range between a few items and 256, but can vary for different applications
4. *Momentum:*
   - Helps accelerate SGD in the relevant direction and dampens oscillations
   - It does this by adding a fraction of the update vector of the past step to the current update vector
5. *Adam (Adaptive Moment Estimation)*
   - Commonly used method that computes adaptive learning rates for each parameter

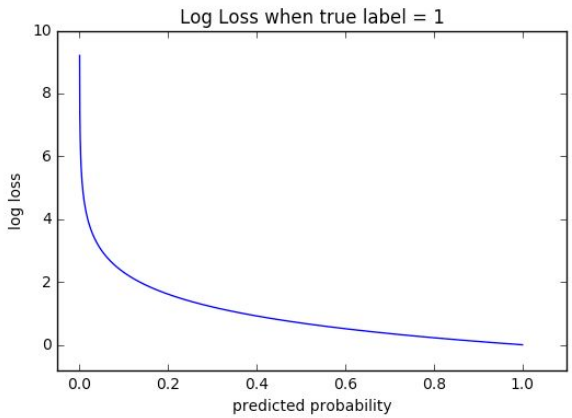
… and many others !!!!

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Loss Function: Cross Entropy

- For classification tasks the *cross entropy* is commonly used in place of the 0-1 loss
- For binary classification: $L(f(\boldsymbol{x}), y) = -y \log\big(f(\boldsymbol{x})\big) - (1-y)\log(1 - f(\boldsymbol{x}))$
- The optimal $f(\boldsymbol{x})$ minimizing this loss function is $f(\boldsymbol{x}) = P(y = 1 \mid \boldsymbol{x})$
  - We are training the neural net output to estimate conditional probabilities
- Note that the expression works if $f(\boldsymbol{x})$ is strictly between 0 and 1
  - An undefined or infinite value would otherwise arise
  - To achieve this, the sigmoid is commonly used as activation for the output layer
- The function is convex

→ Gradient descent (e.g., SGD) works better



Extension to Multi-Class

Label Encoding

Food Name	Categorical #	Calories
Apple	1	95
Chicken	2	231
Broccoli	3	50

→

One Hot Encoding

Apple	Chicken	Broccoli	Calories
1	0	0	95
0	1	0	231
0	0	1	50

state
NY
WA
CA

➡

AL	...	CA	...	NY	...	WA	...	WY
0	...	0	...	1	...	0	...	0
0	...	0	...	0	...	1	...	0
0	...	1	...	0	...	0	...	0

- One-hot encoding
  - Output: vector $\boldsymbol{y}$ with one variable for each class
  - $y_i = 1$ if sample in class i, $y_i = 0$ otherwise
  - Avoid having some classes "closer" to others as when using class index
  - Increases output data dimensionality
- Extension of cross-entropy to multi-class
  - Labels one-hot encoded, vector function $\boldsymbol{f}$ to be estimated
  - $f_i(\boldsymbol{x})$ = estimated probability that $\boldsymbol{x}$ belong to class i

$$L(\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{y}) = -\sum_i y_i \log\left(f_i(\boldsymbol{x})\right)$$

In Practice: Many DL Tools.....

- Many deep learning frameworks
- Supported by large research entities and companies
- Optimized for GPU computing

Tensorflow (Google)

Keras: higher level framework for easier implementation

- Tutorial on Keras in January

Caffe (University of Berkley)

PyTorch (Facebook)

Microsoft Cognitive Toolkit

… and many others